Chapter 4. HOF Domain

4.1 Introduction

A function is *higher-order* if it takes a function as an argument and/or returns a function. A classic example is *mapcar* in Lisp, which takes two arguments, a function and a list. Its behavior is to apply the function to each element of the list and to return a list of the results. The HOF domain implements a similar function, in the form of a star called `Map`, that can apply any other star (or galaxy) to the sequence(s) at its inputs. Many other useful higher-order functions are also provided by this domain.

The HOF domain provides a collection of stars designed to be usable in all other Ptolemy domains. To preserve this generality, not all interesting higher-order functions can be implemented in this domain. As a consequence, some individual domains may also define higher-order functions. In fact, any higher-order function with domain-specific behavior *must* be implemented in its respective domain. The HOF domain is intended to be included as a subdomain by all other domains, although in release 0.6, only SDF, DDF, DE and CGC include the HOF domain. Also, at least in release 0.6, the HOF domain cannot[1] be used in `ptcl`, the textual user interface to Ptolemy. It can only be used within `pigi`, the graphical interface.

A common feature shared by all the stars in this domain is that they perform all of their operations in the `setup` method. Moreover, their basic operation is always to disconnect themselves from the graph in which they appear and then to self-destruct. Since the setup method of the stars in a universe is invoked before the setup method of the scheduler, the scheduler never sees the HOF stars. They will have self-destructed by the time the scheduler is invoked. This is why these stars will work in any domain. In code generation domains, an important feature of the HOF stars is that they add no run-time overhead at all, since they self-destruct before code generation begins, and therefore do not appear in any form in the generated code.

Many of the HOF stars will replace themselves with one or more instances of another star or galaxy, called the *replacement block*. Replacement blocks generally go into the graph in the same position originally occupied by the HOF star, but different HOF stars will connect these replacement blocks in different ways.

Some HOF stars have no replacement block. Before they self destruct, they will typically only alter the connections in the graph without adding any new blocks. An example is the `BusMerge` block, which merges two busses into one wider bus. These stars are called *bus manipulation* stars.

The experienced reader may have some difficulty connecting the concept of higher-order functions, as implemented in this domain, to that used in functional programming. This

1. The HOF domain calls Tk_DoOneEvent() so that if you accidentally specify an infinite recursion (easy to do with HOF), you can hit the STOP button and abort the run. However, Tk is not linked into ptcl, so this call cannot be made. The fix would be to abstract Tk_DoOneEvent so that ptcl can call a dummy function and pigi can call the real function. We hope to fix this in a later release.

issue is covered in some depth in [Lee95], but we can nonetheless give a brief motivation here. In functional languages, there is no syntactic difference between a function argument that is a data value, one that is a stream (an infinite sequence of data values), and one that is a function. In visual programming, however, functions typically have two very different syntaxes for their arguments. Ptolemy is no exception. Stars and galaxies in Ptolemy are functions with two kinds of arguments: input streams and parameters. The HOF domain only contains stars where a parameter may be function. It does not contain any stars that will accept functions at their input portholes as part of an input stream, or produce functions at their output portholes. Although in principle such higher-order functions can be designed in Ptolemy, their behavior would not be independent of their domain, so the HOF domain would be the wrong place for them.

4.2 Using the HOF domain

The HOF stars are found in the main palettes of the domains that use them. For example, the HOF stars used in the SDF domain are found in the main SDF palette. Typically, domains that include the HOF stars will also include demos that use those stars in their demo palette. Thus, the DE demo palette contains a section of Higher Order Function demonstrations. The HOF stars can be used just as if they belonged to the domain in which you are working. Although the examples given below are drawn from the SDF domain, please keep in mind this versatility.

4.2.1 The Map star and its variants

The `Map` star is the most basic of all HOF stars. Its icon is shown below:

It has the following parameters:

blockname	The name of the replacement block.
where_defined	The full path and facet name for the definition of blockname.
parameter_map	How to set the parameters of the replacement block.
input_map	How to connect the inputs.
output_map	How to connect the outputs.

The name of the replacement block is given by the *blockname* parameter. If the replacement block is a galaxy, then the *where_defined* parameter should give the full name (including the full path) of a facet that, when compiled, will define the block. This path name may (and probably should) begin with the environment variable `$PTOLEMY` or ~*username*. This lends a certain immunity to changes in the filesystem organization. Currently, the file specified must be an oct facet, although in the future, other specifications (like `ptcl` files) may be allowed. Usually, the oct facet simply contains the definition of the replacement galaxy. If the replacement block is a built-in star, then there is no need to give a value to the *where_defined* parameter.

The `Map` star replaces itself in the graph with as many instances of the replacement block as needed to satisfy all of the inputs to the Map star. Consider the example shown in figure 4-1. The replacement block is specified to be the built-in `RaisedCosine` star. Since this is built-in, there is no need to specify where it is defined, so the *where_defined* parameter is blank. The `RaisedCosine` star has a single input named *signalIn* and a single output named *signalOut*, so these names are given as the values of the *input_map* and *output_map* parameters. The *parameter_map* parameter specifies the values of the *excessBW* parameter for each instance of the replacement block to be created; *excessBW* specifies the excess bandwidth of the raised cosine pulse generated by the star. The syntax of the *parameter_map* parameter is discussed in detail below, but we can see that the value of the *excessBW* parameter will be 1.0 for the first instance of the `RaisedCosine` star, 0.5 for the second, and 0.33 for the third.

The horizontal slash through the last connection on the right in figure 4-1 is a `Bus`, which is much like a delay in that the icon is placed directly over the arc without any connections. Its single parameter specifies the number of connections that the single wire represents. Here, the bus width has to be three or the `Map` star will issue an error message. This is because there are three inputs to the `Map` star, so three instances of the `RaisedCosine` star will be created. The three outputs from these three instances need somewhere to go. The result of running this system is shown in figure 4-2

The block diagram in figure 4-1 is equivalent to that in figure 4-3. Indeed, once the setup method of the `Map` star has run, the topology of the Ptolemy universe will be exactly as figure 4-3. The `Map` star itself will not appear in the topology, so examining the topology with, for example, the `ptcl print` command will not show a `Map` star instance.

In both figures 4-1 and 4-3, the number of instances of the `RaisedCosine` star is specified graphically. In figure 4-1, it is specified by implication, through the number of instances of the `Impulse` star. In figure 4-3 it is specified directly. Neither of these really takes advantage of higher-order functions. The block diagram in figure 4-4 is equivalent to both 4-1 and 4-3, but can be more easily modified to include more or fewer instances of the `RaisedCosine` star. It is only necessary to modify parameters, not the graphical representa-

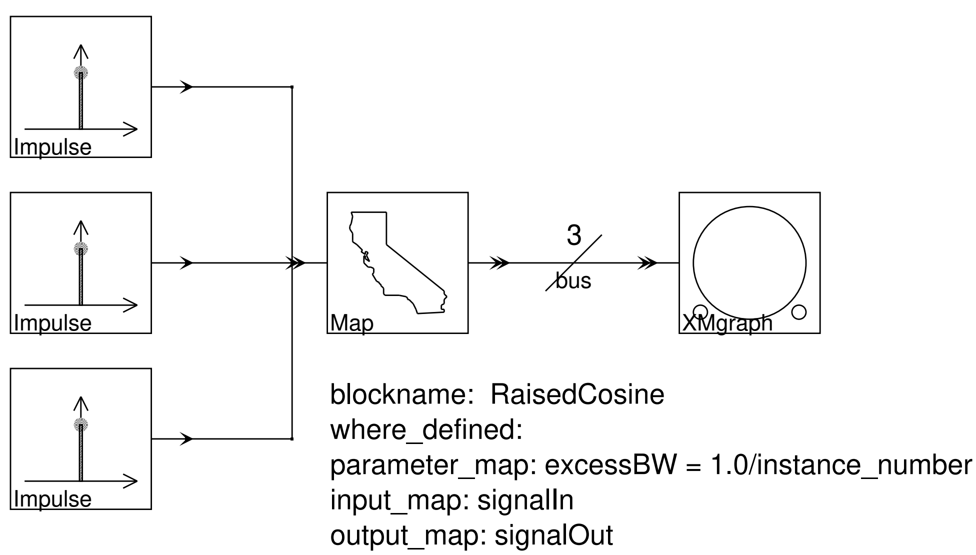


FIGURE 4-1: An example of the use of the `Map` star to plot three different raised cosine pulses.

tion. For example, if the value of the bus parameters in figure 4-4 were changed from 3 to 10, the system would then plot ten raised cosines instead of three.

The left-most star in figure 4-4 is a variant of the `Map` star called `Src`. It has no inputs, and its output data type is specialized to “float” (recall that ports of type “float” in Ptolemy actually communicate data of type “double” in C++). Ptolemy block diagrams are strongly typed, so any star with no inputs must have specific, well-defined types for its outputs (rather than “anytype”). There are variants of the `Src` higher-order star for all built-in data types in Ptolemy. The complete list of higher-order stars is given below.

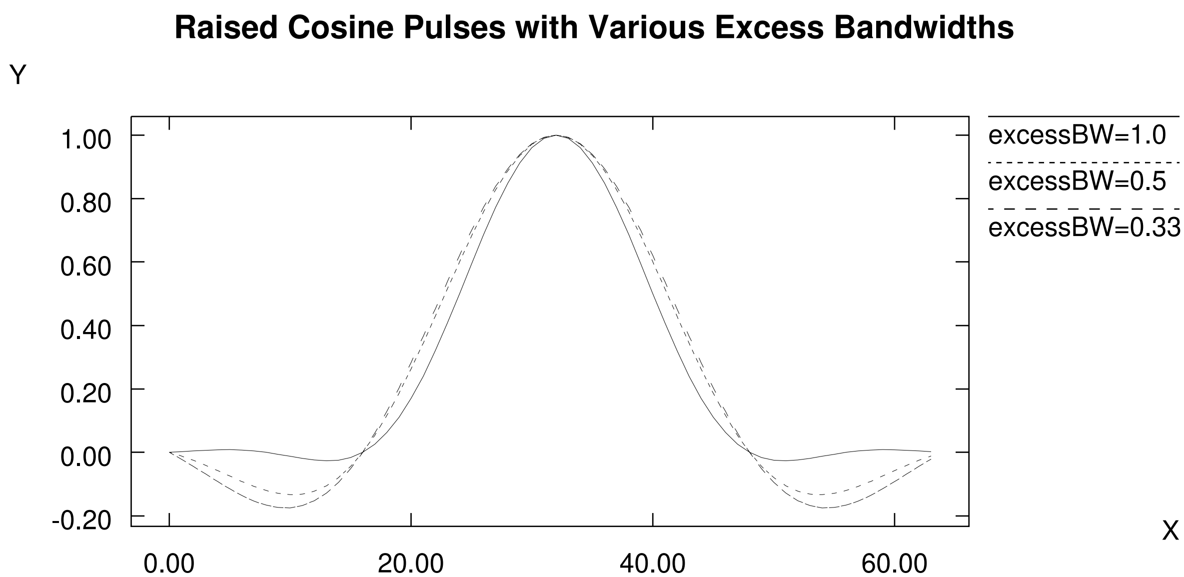


FIGURE 4-2: The plot that results from running the system in figure 4-1.

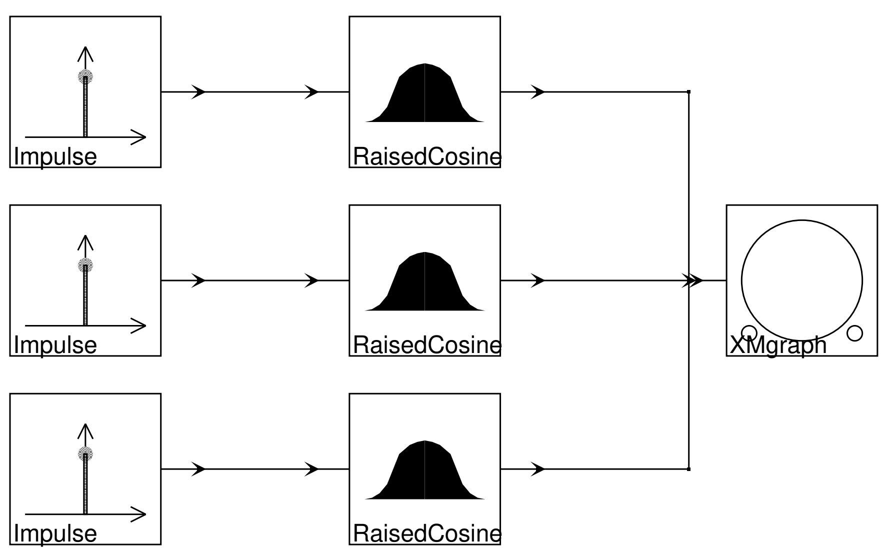
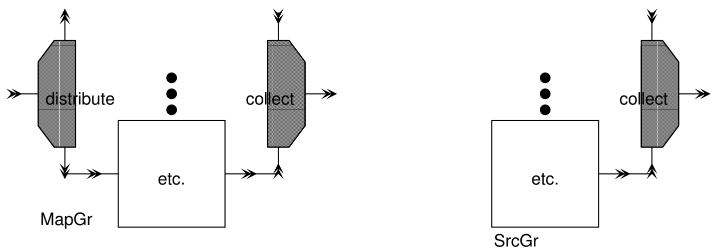


FIGURE 4-3: A block diagram equivalent to that in figure 4-1, but without higher-order functions.

Graphical versions of the Map star

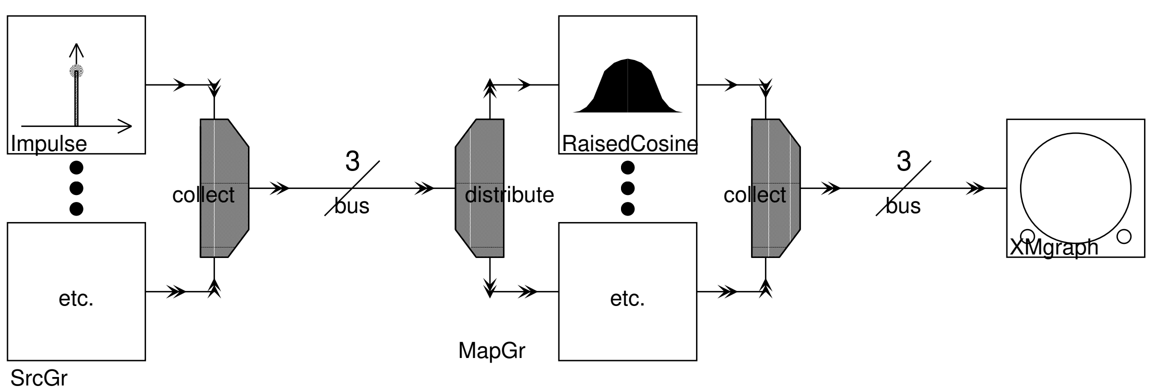
Variants of the `Map` and `Src` stars, called `MapGr` and `SrcGr`, have the following icons:



It is important to realize that `MapGr` and `SrcGr` are single icons, each representing a single star. The complicated shape of the icon is intended to be suggestive of its function when it is found in a block diagram. The `MapGr` and `SrcGr` stars work just like the `Map` and `Src` stars, except that the user specifies the replacement block graphically rather than textually. For example, the system in figure 4-4 can be specified as shown in figure 4-5. Notice that replace-

blockname: Impulse	blockname: RaisedCosine
where_defined:	where_defined:
parameter_map:	parameter_map: excessBW = 1.0/instance_number
output_map: output	input_map: signalIn
	output_map: signalOut

FIGURE 4-4: A block diagram equivalent to that in figures 4-1 and 4-3, except that the number of instances of the `RaisedCosine` and `Impulse` stars can be specified by a parameter.

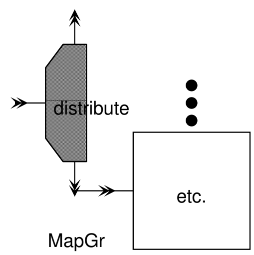


parameter_map: parameter_map: excessBW = 1.0/instance_number

FIGURE 4-5: A block diagram equivalent to that in figure 4-4 except that the replacement blocks for the two higher-order stars are specified graphically rather than textually.

ment blocks `Impulse` and `RaisedCosine` each have one instance wired into the block diagram as an example. Thus, there is no reason for the *blockname*, *where_defined*, *input_map*, or *output_map* parameters. The `MapGr` and `SrcGr` stars have only a single parameter, called *parameter_map*. The syntax for this parameter is the same as for the `Map` star, and is fully explained below.

A variant of the `MapGr` star has the icon shown below:



This version just represents a `MapGr` star with no outputs.

A more complex application of the `MapGr` star is shown in figure 4-6. Here, the replacement block is a `Commutator`, which can take any number of inputs. The bus connected to its input multiporthole determines how many inputs will be used in each instance created by the `MapGr` star. In the example in figure 4-6, it is set to 2. Thus, each instance of the replacement block processes two input streams and produces one output stream. Consequently, the input bus must be twice as wide as the output bus, or the `MapGr` star will issue an error message. This example produces the plot shown in figure 4-7. A key advantage of higher-order functions becomes apparent when we realize that these parameters can be changed. If the parameters are modified to generate 8 instances of the `Commutator` star, then the output plot will be as shown in figure 4-8.

Setting parameter values

The *parameter_map* parameter of the `Map` star and related stars can be used to set parameter values in the replacement blocks. The *parameter_map* is a string array, a list of

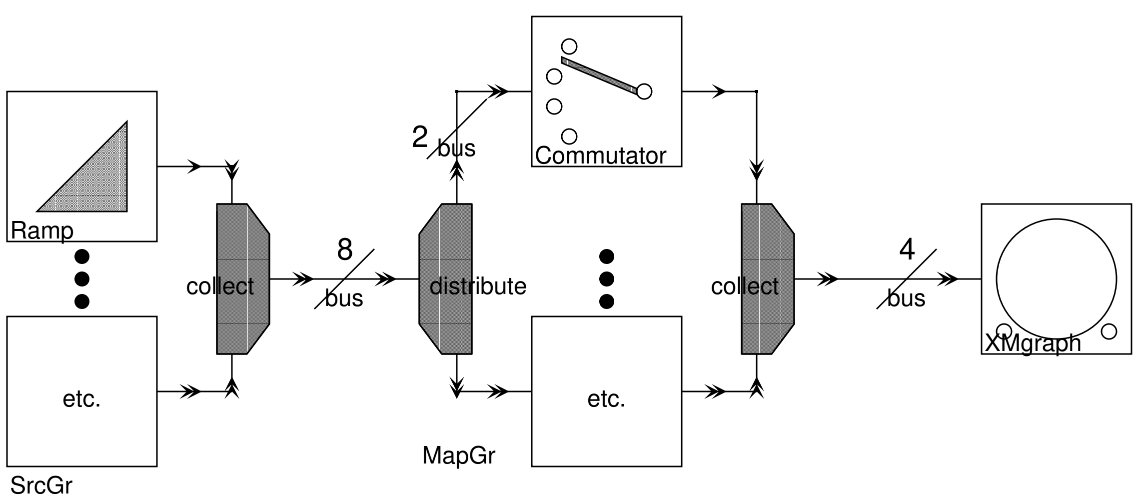


FIGURE 4-6: A more complicated example using higher-order functions.

strings. The strings are in pairs, where the pairs are separated by spaces, and there are four acceptable forms for each pair:

```
name value
name(number) value
name = value
name(number) = value
```

There should be no spaces between *name* and (*number*), and the name cannot contain spaces, =, or (. In all cases, *name* is the name of a parameter in the replacement block. In the first and third cases, the value is applied to all instances of the replacement block. In the second and fourth cases, it is applied only to the instance specified by the instance *number*, (which starts with 1). The third and fourth cases just introduce an optional equal sign, for readability. If the = is used, there must be spaces around it.

The *value* can be any usual Ptolemy expression for giving the value of a parameter. If this expression has spaces in it, however, then the value should appear in quotation marks so that the whole expression is kept together. If the string `instance_number` appears anywhere

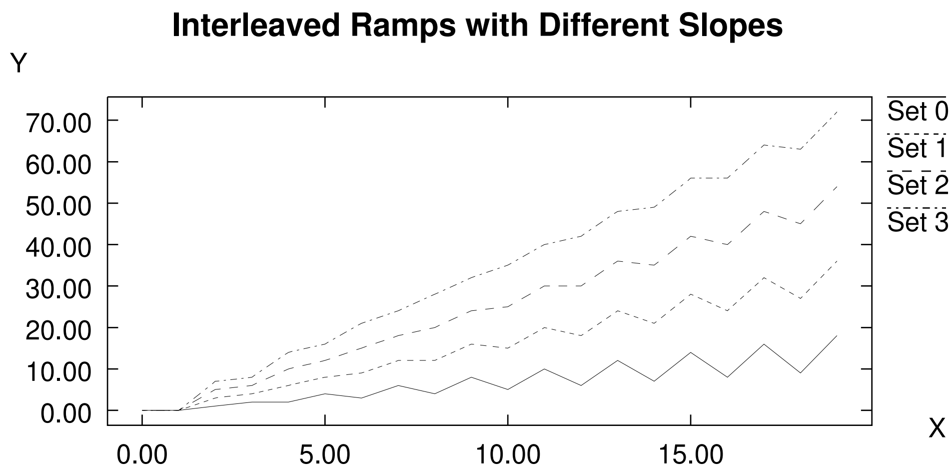


FIGURE 4-7: The plot created by running the system in figure 4-6.

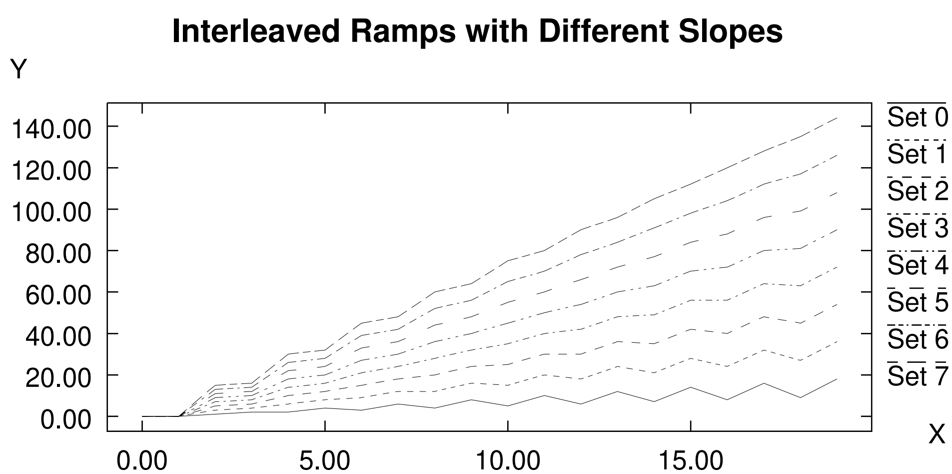


FIGURE 4-8: If the parameters in figure 4-6 are modified to double the number of plots, we get this output.

in *value*, it will be replaced with the instance number of the replacement block. Note that it need not be a separate token. For example, the value `xxxinstance_numberyyy` will become `xxx1yyy` for the first instance, `xxx2yyy` for the second, etc. After all appearances of the string `instance_number` have been replaced, *value* is evaluated using the usual Ptolemy expression evaluator for initializing String Array states.

For example, in figure 4-1, the `Map` star has a *blockname* of `RaisedCosine`, and a *parameter_map* of

```
excessBW = 1.0/instance_number
```

When the system is run, the `Map` star will create three instances of RaisedCosine. The first instance will have its excessBW parameter set to 1.0 (which is 1/1), the second instance of RaisedCosine will have a excessBW of 0.5 (1/2), and the third will have a excessBW of 0.33 (1/3). Since the other RaisedCosine parameters are not mentioned in the parameter_map, they are set to their default values.

As a further example, suppose *parameter_map* of the `Map` star in figure 4-1 were set to

```
excessBW(1) 0.6 excessBW(2) 0.5 excessBW(3) 0.4 length 128
```

The first `RaisedCosine` would then have an *excessBW* of 0.6, the second would have an *excessBW* of 0.5 and the third would have 0.4 for its *excessBW*. All three of the `Raised-Cosine` stars would have a *length* of 128 instead of the default length.

Number of replacement blocks

The number of instances of the replacement block is determined by the number of input or output connections that have been made to the `Map` star. Suppose the `Map` star has M_I inputs and M_O outputs connected to it. Suppose further that the replacement block has B_I input ports and B_O output ports. Then

$$N = \frac{M_I}{B_I} = \frac{M_O}{B_O}$$

is the number of instances that will be created. This must be an integer. Moreover, the number of input and output connections must be compatible (must satisfy the above equality), or you will get an error message like: "too many inputs for the number of outputs."

How the inputs and outputs are connected

The first B_I inputs to the `Map` star will be connected to the inputs of the first instance of the replacement block. To determine in what order these B_I connections should be made, the names of the inputs to the replacement block should be listed in the *input_map* parameter in the order in which they should be connected. There should be exactly B_I names in the *input_map* list. The next B_I inputs to the `Map` star will be connected to the next replacement block, again using the ordering specified in *input_map*. Similarly for the outputs. If there are no inputs at all, then the number of instances is determined by the outputs, and vice versa.

Substituting blocks with multiple input or output ports

When the replacement block has a multiple input port or a multiple output port (shown graphically as a double arrowhead), the name given in the *input_map* parameter should be the name of the multiple port, repeated for however many instances of the port are desired.

For example, the `Add` star has a multiple input port named “input”. If we want the replacement `Add` star(s) to have two inputs each, then *input_map* should be `input input`. If we want three inputs in each replacement block, then input_map should be `input input input`. Note that *input_map* and *output_map* are both of the String Array type. Thus one can use the shortcut string `input[3]` instead of the cumbersome `input input input` string. These two forms are equivalent as `input[3]` is converted automatically to `input input input` when the parameter is initialized by Ptolemy.

A note about data types

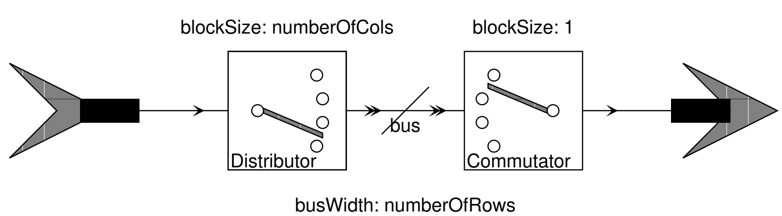
In the `Map` star, the output data type is ANYTYPE, where the specific type will be derived from the type of the input(s). This creates a problem when there are no inputs. Thus, the zero-input version of this star is implemented as a family of derived stars called `Src` stars, each of which has a specific output data type.

There are currently other problems with this mechanism, in that when type resolution is done before the block substitution occurs, there is no information about the substitution block. It is best, therefore, when using the `Map` star, to make all type conversions explicit. Future versions of Ptolemy may implement type resolution differently, hopefully overcoming these problems. The problems do not occur with the `MapGr` stars, because these are able to determine the data type of the inputs and outputs from the sample replacement block, which is sure to be present when type resolution is completed.

4.2.2 Managing multidimensional data

There are many alternatives in Ptolemy for managing multidimensional data. One simple possibility is to use the `MatrixParticle` class. This encapsulates a matrix into a single particle. Another alternative is to use the `Message` class to define your own multidimensional data structure. A third alternative (which is still highly experimental) is to use the multidimensional synchronous dataflow (`MDSDF`) domain. A fourth alternative, discussed here, is to embed your multidimensional data into one-dimensional streams. Higher-order functions become extremely useful in this case. Our discussion will center on using the SDF domain, although the same principles could be applied in other domains as well.

A two-dimensional array of data can be embedded in a one-dimensional stream by rasterizing it. This means that the sequence in the stream consists of the first row first, followed by the second row, followed by the third, etc. This is one example of a *multiprojection*, so called because higher-dimensional data is projected onto a one-dimensional sequence. Typically, however, we wish to perform some operations row-wise, and others column-wise, so the rasterized format can prove inconvenient. Row-wise operations are easy if the data is rasterized, but column-wise operations are awkward. Fortunately, in the SDF domain, we can transpose the data with a cascade of two stars, a `Distributor` and a `Commutator`, as shown below:



If the input is row-wise rasterized, then the output will be column-wise rasterized, meaning that the first column will come out first, then the second column, then the third, etc. It has been shown that any transposition of any arbitrary multiprojection can be accomplished with such a cascade [Khi94].

As an example of the use of a multi-projection transformation, consider the two-dimensional FFT shown in figure 4-9. Recall that a two-dimensional discrete Fourier transform can be implemented by first applying a one-dimensional DFT to the rows and then applying a one-dimensional DFT to the columns. The system in figure 4-9 does exactly this. To see how it works, recall that the `FFTCx` star in the SDF domain has two key parameters, the *order* and the *size*. The *size* is the number of input samples read each time the FFT is computed. For the row FFT, this should be equal to the number of columns. These samples are then padded with zeros (if necessary) to get a total of 2^{order} samples. The `FFTCx` star then computes a 2^{order} point FFT, producing 2^{order} complex outputs. In figure 4-9, these outputs are then transposed so that they are column-wise rasterized. The second `FFTCx` star then computes the FFT of the columns. The output is column-wise rasterized.

The effect of a transposition can be accomplished using higher-order functions in a way that is sometimes more intuitive. In particular, a matrix can be represented as a bus, where each connection in the bus carries one row, and the width of the bus is the number of rows. To make this concrete, the same two-dimensional FFT is re-implemented in figure 4-10 using this representation. The rasterized input is first converted into a bus, where each connec-

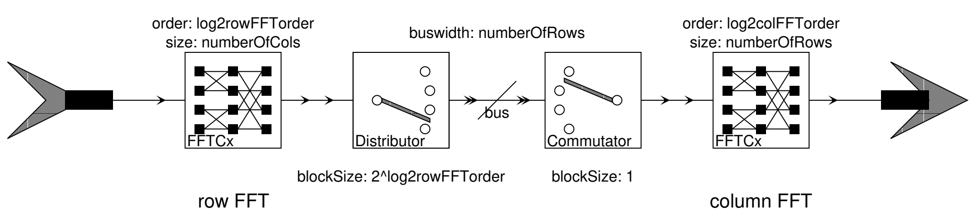


FIGURE 4-9: A two-dimensional FFT operating on rasterized input data and using the `Distributor` and `Commutator` stars to transpose one multiprojection into another.

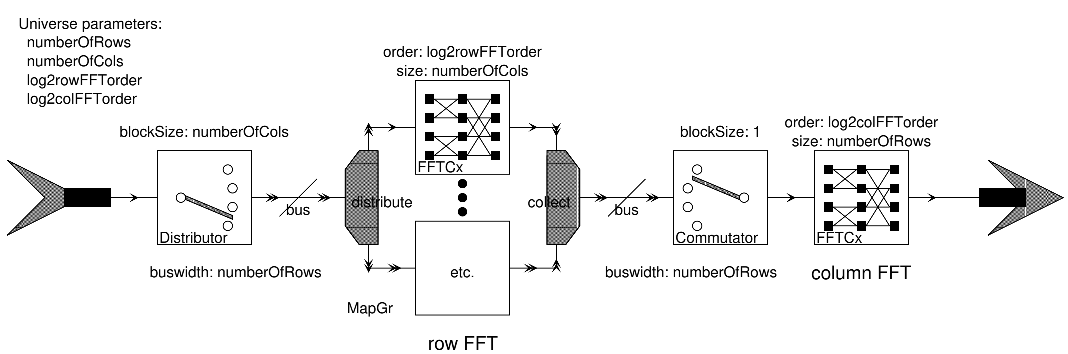


FIGURE 4-10: A two-dimensional FFT implemented using a higher-order function for the row FFTs.

tion in the bus represents one row. The `MapGr` star is then used to apply the FFT to each signal in the bus. The results are recombined in column-wise rasterized format, and the column FFT is computed.

These examples are meant to illustrate the richness of possibilities for manipulating data. A more complete discussion of these issues and their application to radar signal processing are given in [Khi94].

4.2.3 Other higher-order control structures

The `Map` star and its variants apply instances of their replacement block in parallel to the set of input streams. Another alternative is provided by the `Chain` star, which strings together some specified number of instances of the replacement block in series. The parameters are similar to those of the `Map` star, except for the addition of *internal_map*. The *internal_map* parameter specifies connections made between successive instances of the replacement block in the cascade. It should consist of an alternating list of output and input names for the replacement block.

An example of the use of the `Chain` star is a string of biquad filters in series. The `IIR` filter star, which can be used to create a biquad filter, has an input named “signalIn” and an output named “signalOut.” To have a string of these stars in series, one would want the output of the first `IIR` star in the series to be connected to the input of the second star. And the output of the second star should be connected to the input of the third, etc. Thus, a `Chain` star that is a series of biquad filters would have an *internal_map* of

```
signalOut signalIn
```

to specify that the output of one block is connected to the input of the next.

Another variant is the `IfElse` block. This star is just like `Map`, except that it has two possible replacement blocks. If the *condition* parameter is `TRUE`, then the *true_block* is used. Otherwise, the *false_block* is used. It is important to realize that the *condition* parameter is evaluated at setup time. Once a replacement block has been selected, it cannot be changed. There are two uses for this block. It can be used to parameterize a galaxy in such a way that the parameter determines which of two functions is used within the computation. More interestingly, it can be used to implement statically-evaluated recursion.

4.2.4 Statically evaluated recursion

The `Map` star and its variants replace themselves with an instance of the block specified as the replacement block. What if that block is a galaxy within which the very `Map` star in question sits? This is a recursive reference to the galaxy, but a rather awkward one. In fact, in such a configuration, the setup phase of execution will never terminate. The user has to manually abort such an execution in order to get it to terminate.

The `IfElse` star, however, can conditionally specify one of two replacement blocks. The *condition* parameter determines which block. One of the two replacement blocks can be a recursive reference to a galaxy as long as the *condition* parameter is modified. When the condition parameter changes state, going from `TRUE` to `FALSE` or `FALSE` to `TRUE`, then the choice of replacement block inside the new galaxy instance will change. This can be used to terminate the recursion.

Consider the example shown in figure 4-11. This galaxy has a single parameter, *log2framesize*. It will read $2^{log2framesize}$ input particles and rearrange them in bit reversed order. That is, they will emerge from the galaxy as if their binary address had been interpreted with the high-order bit reinterpreted as a low-order bit, and vice versa. Suppose for example that *log2framesize* = 3, and that the input sequence is 10,11,12,13,14,15,16,17. Then the output sequence[1] will be 10,14,12,16,11,15,13,17. To accomplish this, the *bit_reverse* galaxy uses two `IfElse` stars, each with a conditional recursive reference to the *bit_reverse* galaxy. The condition is *log2framesize*-1, and the *log2framesize* parameter for the inside instances of the galaxy is set to *log2framesize*-1. When *log2framesize* gets to zero, the replacement block becomes a `Gain` star with unity gain (which of course has no effect). This terminates the recursion.

With *log2framesize* = 3, after the setup phase, the topology of the galaxy will have become that shown in figure 4-12. It is much easier to see by inspection of this topology how the bit reversal addressing is accomplished. It is also easier to see how this operation could be made more efficient (the innermost cluster of `Distributors`, `Gains`, and `Commutators` has no effect at all). Unfortunately, we currently have no mechanism for automatically displaying this expanded graph visually. It can, however, be examined using `ptcl`.

The *bit_reverse* galaxy performs the sort of data manipulation that is at the heart of the decimation-in-time FFT algorithm. See [Lee94] for an implementation of that algorithm using these same techniques (or see the demos).

4.2.5 Bus manipulation stars

One consequence of the introduction of higher-order functions into Ptolemy is that busses have suddenly become much more useful than they used to be. Recall that the bus icon resembles a diagonal slash, as shown in figure 4-1, and is placed over a connection, much like

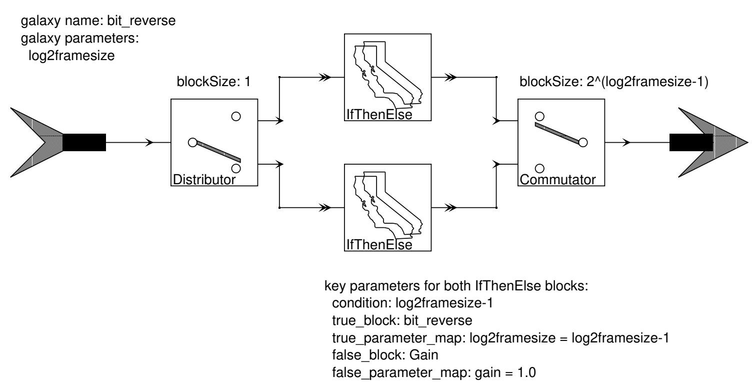


FIGURE 4-11: A recursive galaxy, where the `IfElse` HOF star replaces itself with an instance of the same galaxy until its *condition* parameter gets to zero.

1. For those unfamiliar with bit-reversed addressing, here is a quick introduction. Since *log2framesize* is 3, galaxy will read in 2^3=8 values at a time. The first value (10) has address 0 (since computers always seem to count from zero) which is 000 in binary. Reversed, its address is still 000 so it is output first. The second value (11) has address 1 which is 001 in binary. Reversed, its address is 100 binary which is 4. Thus the value (11) is output in the fifth spot. As a final example, the seventh value (16) has address 6 which is 110 in binary. Reversed, its binary value is 011 which is 3 and the value (16) is output forth. After the first 8 values are read, the cycle is repeated for the next 8 values.

a delay. Its single parameter specifies the width of the bus.

Fortunately, while increasing the demand for busses, higher-order functions also provide a cost effective way to manipulate busses. Like the `Map` star and its variants, the bus manipulation stars in the HOF domain modify the graph at setup time and then self-destruct. Thus, they can operate in any domain, and they introduce no run-time overhead.

An example of the use of the `BusSplit` star is shown in figure 4-13. A bank of 12 random number generators produces its output on a bus. The bus is then split into two busses of width 6 so that subsets of the signals can be displayed together. The `BusSplit` star rewires the graph at setup time and then self-destructs. Thus, it introduces zero run-time overhead.

A more interesting bus manipulation star is the `Nop` star, so called because it really

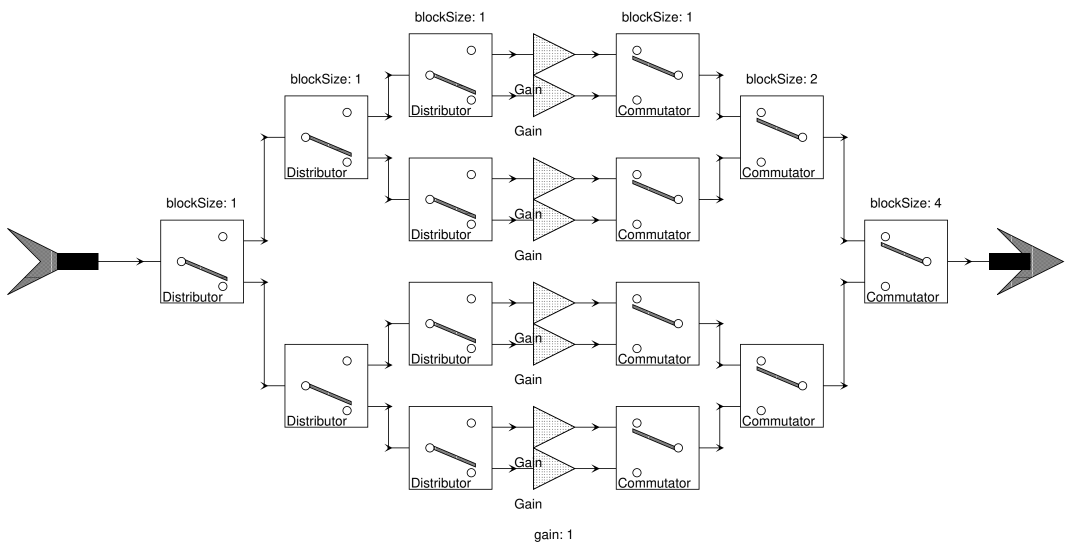


FIGURE 4-12: An expansion of the graph in figure 4-11, representing its topology after the setu phase assuming *log2blocksize* at the top level is 3.

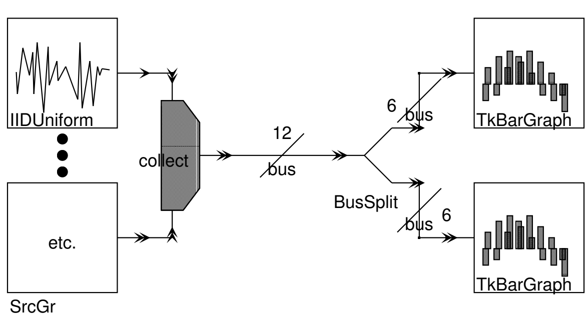
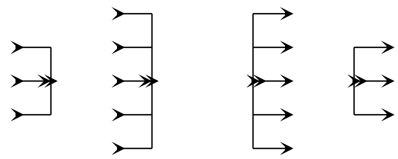


FIGURE 4-13: A BusSplit star is used to divide a set of signals into two subsets for separate display.

performs no function at all. It can have any number of inputs, but the number of outputs must be the same as the number of inputs. All it does is connect its inputs to its outputs (at setup time) and the self-destruct. It has many icons, four of which are shown below:



The icon on the left has three individual input ports, and simply combines them into an output multiporthole. This multiporthole would normally be connected to a bus, which must be of width three. Thus, this icon provides a way to create a bus from individual connections. The next icon is similar, except that it has five input lines. The next two icons do the reverse. They are used to break out a bus into its individual components.

Examples of the uses of `Nop` stars are shown in figure 4-14. Three signals are individually generated at the left by three different source stars. These signals are then combined into a bus of width three using a `Nop` star. The bus is then broken out into three individual lines, which are fed to three `Gain` stars. The most interesting use of the `Nop` star, however, is the one on the right. The `XMgraph` star shown there has a multiporthole input. The `Nop` star is simply deposited on top of the multiporthole to provide it with three individual inputs. Why do this? Because when connecting multiple signals to a multiporthole input, as done for example in figure 4-3, it is difficult to control which input line goes to which specific porthole in the multiporthole set. Putting the `Nop` star on the porthole gives us this control with no additional runtime cost.

Recall that many stars in Ptolemy that have multiportholes have multiple icons, each icon configured with a different number of individual ports. This proliferation of icons is no longer necessary, and these icons will disappear from the palettes in future versions of Ptolemy. This will considerably reduce clutter in the Ptolemy palettes.

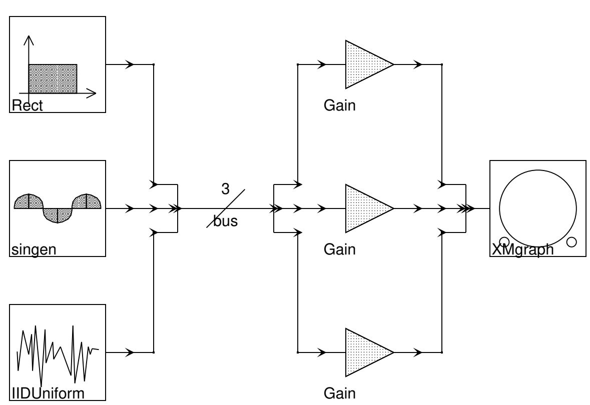


FIGURE 4-14: The Nop star is used to create busses from individual connections, to break busses down into individual lines, and to break out multiportholes into individual ports.

4.3 An overview of the HOF stars

The Higher Order Function stars are accessed through the main palette of the domains that support HOF. For example, the HOF stars are a sub-palette of the SDF star palette since the SDF domain supports HOF. The top-level palette for the HOF domain is shown in figure 4-15.

4.3.1 Bus manipulation stars

The top group in the main HOF palette are the bus manipulation stars, summarized below:

`BusMerge` Bridge inputs to outputs and then self-destruct. This star merges two input busses into a single bus. If the input bus widths are M_1 and M_2, and the output bus width is N, then we require that $N = M_1 + M_2$. The first M_1 outputs come from the first input bus, while the next M_2 outputs come from the second input bus.

`BusSplit` Bridge inputs to outputs and then self-destruct. This star splits an input bus into two. If the input bus width is N, and the output bus widths are M_1 and M_2, then we require that $N = M_1 + M_2$. The first M_1 inputs go the first output bus, while the next M_2 inputs go to the second output bus.

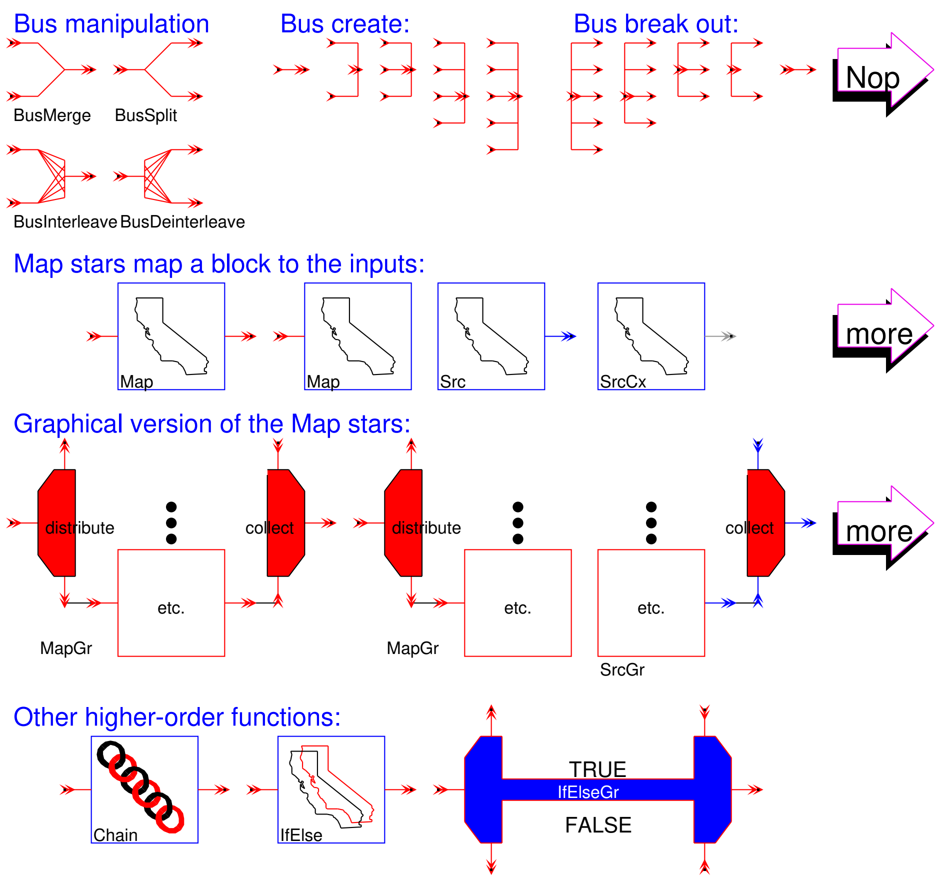


FIGURE 4-15: The top-level palette for the higher-order function stars. The icons labeled “more” point to supplementary palettes.

`BusInterleave` Bridge inputs to outputs and then self-destruct. This star interleaves two input busses onto a single bus. The two input busses must have the same width, which must be half the width of the output bus. The input signals are connected to the output in an alternating fashion.

`BusDeinterleave` Bridge inputs to outputs and then self-destruct. This star deinterleaves a bus, producing two output busses of equal width. The input bus must have even width. The even numbered input signals are connected to the first output bus, while the odd numbered input signals are connected to the second output bus.

`Nop` Bridge inputs to outputs and then self-destruct. This star is used to split a bus into individual lines or combine individual lines into a bus. It is also used to break out multi-inputs and multi-outputs into individual ports. These icons are labeled “`BusCreate`” and “`BusSplit`”, suggesting their usual function.

If you look inside the icon labeled “Nop” to the right of the above stars, you will open another palette with more icons for the `Nop` stars, shown in figure 4-16.

4.3.2 Map-like stars

The icons for the less frequently used versions of these stars can be found by looking inside the icons labeled “more” in the main HOF palette (figure 4-15).

`Map` Map one or more instances of the named block to the input stream(s) to produce the output stream(s). This is implemented by replacing the `Map` star with one or more instances of the named block at setup time. The replacement block(s) are connected as specified by *input_map* and *output_map*, using the existing connections to the `Map` star. Their parameters are determined by *parameter_map*. See “Setting parameter values” on

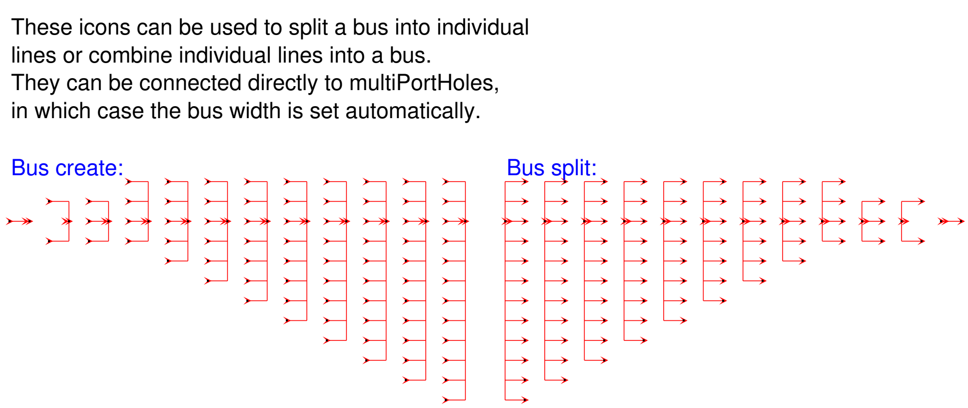


FIGURE 4-16: A secondary palette with a more complete set of icons for the Nop star. This palette is accessed by looking inside the icon labeled “Nop” in figure 4-15.

page 4-152 for examples of the use of *parameter_map*.

`Src`	This is identical to the `Map` star, except that the replacement block is a source block (it has no inputs) and the output data type is float.
`SrcCx`	Complex version of `Src`.
`SrcInt`	Integer version of `Src`.
`SrcFix`	Fixed-point version of `Src`.
`Src_M`	`FloatMatrix` version of `Src`.
`SrcCx_M`	`ComplexMatrix` version of `Src`.
`SrcInt_M`	`IntMatrix` version of `Src`.
`SrcFix_M`	`FixMatrix` version of `Src`.
`MapGr`	A variant of the `Map` star where the replacement block is specified by graphically connecting it. There must be exactly one block connected in the position of the replacement block. The `Nop` stars are the only exception: they may be used in addition to the one replacement block in order to control the order of connection.
`SrcGr`	This is identical to the `MapGr` star, except that the replacement block is a source block (it has no inputs) and the output data type is float.
`SrcGrCx`	Complex version of `SrcGr`.
`SrcGrInt`	Integer version of `SrcGr`.
`SrcGrFix`	Fixed-point version of `SrcGr`.
`SrcGr_M`	`FloatMatrix` version of `SrcGr`.
`SrcGrCx_M`	`ComplexMatrix` version of `SrcGr`.
`SrcGrInt_M`	`IntMatrix` version of `SrcGr`.
`SrcGrFix_M`	`FixMatrix` version of `SrcGr`.
`Chain`	Create one or more instances of the named block connected in a chain. This is implemented by replacing the `Chain` star with instances of the named blocks at setup time. The replacement block(s) are connected as specified by *input_map*, *internal_map*, and *output_map*. Their parameters are determined by *parameter_map*. If *pipeline* is `YES`, then a unit delay is put on all internal connections.
`IfElse`	This star is just like `Map`, except that it chooses one of two named blocks to replace itself. If the *condition* parameter is `TRUE`, then the *true_block* is used. Otherwise, the *false_block* is used. This can be used to parameterize the use of a given block,

or, more interestingly, for statically evaluated recursion.

`IfElseGr` A variant of the `IfElse` star where the two possible replacement blocks are specified graphically rather than textually. There must be exactly two blocks connected in the positions of the replacement blocks. The `Nop` stars are the only exception: they may be used in addition to the two replacement blocks in order to control the order of connection. As of this writing, this star cannot be used with recursion.

4.4 An overview of HOF demos

The HOF demos are divided by domain, and are accessed through the demo palette of the individual domain. As of this writing, only the SDF, DDF, DE, and CGC domains have HOF demo palettes.

4.4.1 HOF demos in the SDF domain

The top-level demo palette for the HOF/SDF demos is shown in figure 4-17. The icon labeled "test" points to a set of demos that are not documented here and are used as part of the regression tests in Ptolemy.

`addingSinWaves`

This demo generates a number of sine waves given by the parameter *number_of_sine_waves* and adds them all together. The amplitude of each sine wave is controlled by a Tk slider that is inserted into the control panel when the system is run. The frequency in radians of each sine wave (relative to a sample rate of 2π) is *instance_number* multiplied by $\pi/32$. Thus, the first sine wave will have a period of 64 samples. The second will have a period of 32. The third will have a period of 16, etc. The sum of these sine waves is displayed in bar-graph form.

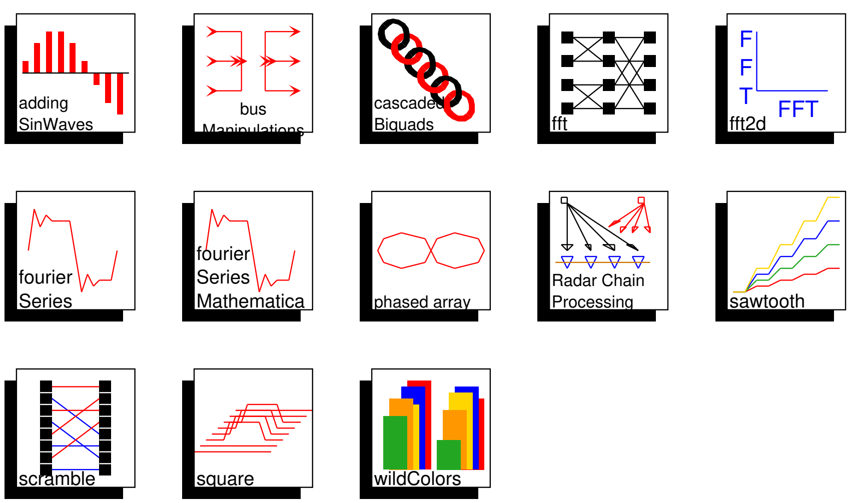


FIGURE 4-17: The top level palette of the HOF demos in the SDF domain.

`busManipulations`
: This demo is shown above in figure 4-14 and explained in the accompanying text.

`cascadedBiquads`
: The `Chain` HOF star is used to construct a cascade of two second-order direct-form recursive filters (biquads). The frequency response of the cascade is compared against the frequency response of a direct-form fourth-order filter with the same transfer function.

`fft`
: This system implements a recursive definition of a decimation-in-time fast Fourier transform, comparing its output against that of a direct implementation in C++. The system is configured to use 32 point FFTs to implement a 256 point FFT. The granularity is controllable with the parameters, and can be taken all the way down to the level of multipliers and adders. This system is discussed in detail in [Lee94].

`fft2d`
: This system generates the same square as in the `square` demo, and then computes its two-dimensional FFT using the method given in figure 4-10.

`fourierSeries`
: This system generates a number of sinusoids as given by the *number_of_terms* parameter. These are then weighted by the appropriate Fourier series coefficients so that the sum of the sinusoids gives the finite Fourier series approximation for a square wave with period given by the *period* parameter.

`fourierSeriesMma`
: This system is similar to the `fourierSeries` system above, but uses Mathematica to calculate parameter values. Mathematica must be licensed on the local workstation for this demo to run.

`phased_array`
: This system models a planar array of sensors with beamforming and steering, such as might be used with a microphone array or a radar system. The sensors can be positioned arbitrarily in a plane. With the default parameters, 16 sensors are uniformly spaced along the vertical axis, half a wavelength apart, except for one, the fourth, which is offset along the horizontal axis by one tenth of a wavelength. The gain of the array as a function of direction is plotted in both polar and rectangular form (the latter in dB). A Hamming window is applied to the sensor data, as is a steering vector which directs the beam downwards. Zoom into the center of the polar plot to see the effect of the offset sensor. Try changing the *parameter_map* of the left-most `MapGr` higher-order function to realign the offset sensor, and observe the effect on the gain pattern.

`RadarChainProcessing`
: This system simulates radar without beamforming. In this simulation, we simulate the effect of an electromagnetic signal traveling from a transmitter to targets and going back to receivers.The delay of the returned signal is used to provide information on the range of the target. The frequency shift, or Doppler effect, is used to provide information on the speed of the target.

Thus, with these parameters, we estimate the target's properties as in a narrow band radar.
The system has been converted from a data parallel form that uses a five-dimensional data array to a functional parallel form that uses higher-order functions to produce streams of streams. The five dimensions are range bin, doppler filters, number of sensors, number of targets and number of pulses. For more information, see `http://ptolemy.eecs.berkeley.edu/papers/Radarsimu.ps.Z`.

`sawtooth` This demo is shown above in figure 4-6 and explained in the accompanying text.

`scramble` This system demonstrates the *bit_reverse* galaxy shown above in figure 4-11 and explained in the accompanying text.

`square` This system demonstrates the `BusMerge` HOF star. It generates an image consisting of a light square on a dark background. The image is first represented using a bus, where each connection in the bus represents one row. The `Commutator` star then rasterizes the image.

`wildColors` This demo is shown above in figure 4-13 and explained in the accompanying text.

4.4.2 HOF demos in the DE domain

At this time, there are only two simple demos in the DE domain.

`poisson` This system generates any number of Poisson processes (default 10) and displays them together. To distinguish them, each process produces events with a distinct value.

`exponential` Combine a number of Poisson processes and show that the interarrival times are exponentially distributed by plotting a histogram. Notice that the histogram bin centered at zero is actually only half as a wide as the others (since the interarrival time cannot be negative), so the histogram displays a value for the zero bin that is half as high as what would be expected.

4.4.3 HOF demos in the CGC domain

The top-level demo palette for the HOF demos in the C Code generation domain (CGC) is shown in figure 4-18.

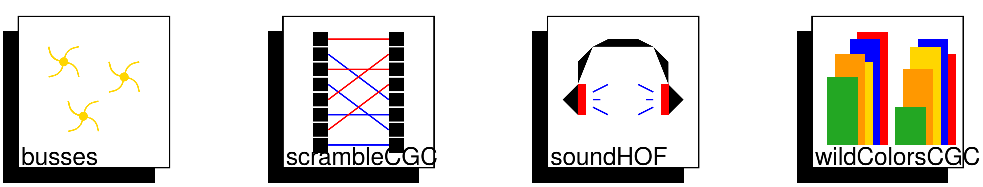


FIGURE 4-18: The top level palette of the HOF demos in the CGC domain.

`busses` Create a set of ramps of different slopes and display them in both a bar chart and using pxgraph.

`scrambledCGC` This system demonstrates recursion in code generation by taking a ramp in and reordering samples in bit-reversed order.

`soundHOF` This system produces a sound made by adding a fundamental and its harmonics in amounts controlled by sliders. This demo will work only on Sun workstations.

`wildColorsCGC` This system is a CGC version of the SDF demo `wildColors`. It creates a number of random sequences and plots them in a pair of bar graphs.